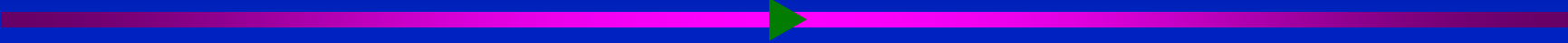


C语言中级培训



八、再谈指针

先说地址的概念

内存地址：物理内存中存储单元的编号。

使用内存：在地址所标识的存储单元中存放数据。

注意：内存单元的地址与内存单元中的数据是两个完全不同的概念。

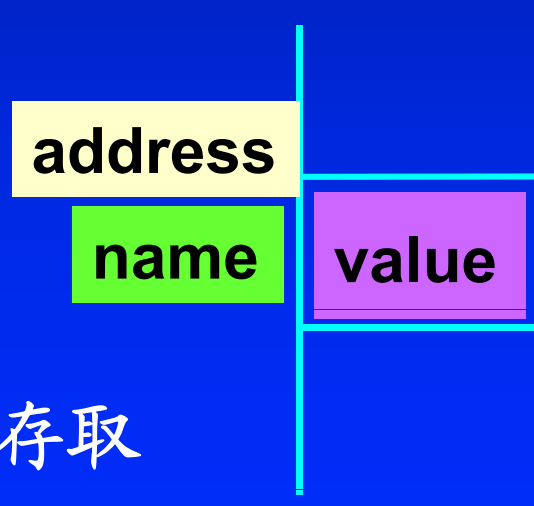
一个是address;

另一个是value;

访问方式：

(1) 直接访问——使用变量名进行存取

(2) 间接访问——通过该变量的地址来访问



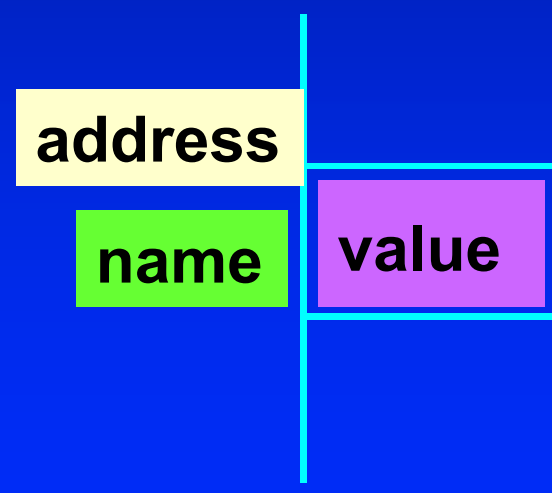
指针的概念

指针是C语言提供给编程者符号化使用硬件地址的一种方法，也是将常量地址变量化的方法。

是一种特殊形式的变量。

特在哪儿？

- 其**value** 是别人的地址、
- 类型不用于指导开辟空间、
- 可以无类型、
- 间址访问才有意义、
- 行为受限。



指针和地址

指针是一种容纳地址的变量，通常也叫指针变量，统称指针。而地址则是内存单元的编号，是值。指针绝对不等同于地址，千万不要把二者意义混淆。

- 实际上指针是数据类型的一种，它应该跟整型、字符型，浮点型是一样的，只不过是组合类型罢了。
- 指针变量只是存储地址类型的变量，它不是数据类型。

指针能够进行加减法，原因并不是因为它是指针。加减法不是指针变量的专利，而是地址这种数据类型的本能，正是因为地址具有加减的能力，所以才使指针作为存放地址的变量能够进行加减运算。

指针的初始化

一般形式:

[存储类型] 数据类型 *指针变量名=初始地址值;

```
例 void main( )  
  {  
    int i;  
    static int *p=&i;    // (×)  
    .....  
  }
```

不能用auto变量的地址
去初始化static型指针
Why ?

指针的声明

b是指针吗?

1) char * a,b;

2) char *a,b;

应该写成:

char *a, *b;

或干脆另写一句: char * b;

这会万无一失!

还可以: typedef char * pchar;

然后 pchar a,b,c;

指针的类型

对于 `int * p = &i` ; 语句的正确理解是：定义了一个类型为 `int *` 的指针。

此时不要将 `int` 和 `*` 分开，因为 `int *` 是个新类型，是组合类型。许多人错误地认为这是定义了一个整型 (`int`) 的指针 (`*p`)。这恰是将“定义了一个指针”，还是“对 `p` 作求值运算”混淆的根源。

```
typedef int * IntPtr;  
IntPtr pa,pb,pc;  
typedef int ** IntPtrPtr;  
IntPtrPtr ppa,ppb,ppc;  
typedef char * CharPtr;  
CharPtr pCh1 , pCh2;  
typedef void * VoidPtr;  
VoidPtr pVoid;
```

指针的使用

可以有以下9种用法:

- 赋值;
- 求值(间接访问);
- 取址;
- 自增/自减;
- 加/减一个整数;
- 求差;
- 比较;
- 作函数的参数或返回值;
- 调用所指的函数。

指针的赋值

仅有5种形式:

p = &a;

p = A;

p = q;

p = f;

p = NULL;

指针是最“手眼通天、神通广大的特权者”，它不受栈的约束，可以访问栈存储区、静态存储区、堆存储区，甚至代码区。

最能体现C的灵活性，但也为C程序埋下了隐患。

指针的算术运算

可以进行的算术运算，只有以下几种：

$px \pm n$, $++px/px++$, $--px/px--$, $px-py$

$px \pm n$: 将指针从当前位置向前 ($+n$) 或后退 ($-n$) n 个数据单位，而不是 n 个字节。显然， $++px/px++$ 和 $--px/px--$ 是 $px \pm n$ 的特例 ($n=1$)。

$px-py$: 两指针之间的数据个数，而不是指针的地址之差。

指出下面各题的错误，分析错误原因

1. `int i,*p;`

`p=i;`

2. `char *p,*q,*r,*t;`

`r=(p+q)/2;`

`t=p+(q-p)/2;`

3. `int *ipt;`

`float *fpt;`

`int(ipt-fpt)>0`

`fpt=ipt;`

指针与变量的关系

指针变量和它所指向的变量之间的关系是用指针运算符“*”表示的。——求值，又叫“间接访问”、“间址运算”

例如，指针变量num_pointer与它所指向的变量num的关系，表示为：*num_pointer，
即*num_pointer等价于变量num。

因此，下面两个语句的作用相同：

num=3;	/*将3直接赋给变量num*/
num_pointer=#	//使num_pointer指向num
*num_pointer=3;	//将3赋给指针所指向的变量

指针变量与零值比较:

指针变量的零值是“空”（记为**NULL**）。尽管**NULL**的值与**0**相同，但是两者意义不同。假设指针变量的名字为**p**，它与零值比较的标准**if**语句如下：

if (p == NULL) // **p**与**NULL**比较，强调**p**是指针变量
或 **if (p != NULL)**

不要写成

if (p == 0) // 容易让人误解**p**是整型变量
if (p != 0)

或：

if (p) // 容易让人误解**p**是布尔变量
if (!p)

有时候我们可能会看到:

if (NULL == p)

为何有意把p和NULL的位置颠倒?

因为当把 **if (p == NULL)**不小心误写成 **if (p = NULL)**时, 编译器会认为 **if (p = NULL)** 是合法的, 它将执行一条赋值语句, 然后将表达式的值作为if的判断依据。显然不是我们的本意。但若把 **if (NULL == p)** 写成 **if (NULL = p)**, 编译器会指出该语法错误, 因为 **NULL**不能被赋值。这样可以避免不易察觉的错误的发生, 让编译器替我们排除更多的错误。

指针的漏洞

VC++6.0竟然允许这样使用指针:

```
#include <stdio.h>
```

```
void main()
```

```
{
```

```
    int a = 1234;
```

```
        // 0x0012ff7c是&a的值。
```

```
    int *p =( (int*)0x0012ff7c );
```

```
    printf("&a = %x\n",&a);
```

```
    printf("a = %d\n*p = %d\n",a,*p);
```

```
}
```

思考会引起什
么严重后果?

!

关于“野指针”

“野指针”是未指向具体变量的指针（又称“指针悬空”，此时它会乱指一气）。

因为创建一个指针时，系统只是分配了指针本身的空间，没有分配所指目标的空间。

“野指针”产生的3种原因：

- 指针变量没有被初始化。指针变量刚被创建时不会自动成为**NULL**指针，它的默认值是随机的；
- 指针被**free**或**delete**之后，没有置为**NULL**，让人误以为它仍然是个合法的指针；
- 指针操作超越了变量的作用范围；

避免“野指针”的对策：

使用指针前一定要保证它指向了有效的内存空间（或者申请，或者让指针指向一块合法的空间）

用**malloc**或**new**申请内存之后，应该立即检查指针值是否为**NULL**。

不要忘记为数组和动态内存赋初值。防止将未被初始化的内存作为右值使用。

避免数组或指针的下标越界，特别要当心发生“多1”或者“少1”操作。

动态内存的申请与释放必须配对，防止内存泄漏。

用**free**或**delete**释放了内存之后，立即将指针设置为**NULL**，防止产生“野指针”。

下面程序会是何结果?——野指针的害处

```
void main()
{
    char *input1, *input2;
    input1 = (char*)malloc(20);
    strcpy (input1, "this is string1");
    printf("%s\n", input1); //
    free(input1);           // 此时的指针是野指针
    input2 = (char*)malloc(20); //input2得到的空间是 input1的
    strcpy (input2, "this is string2");
    printf("%s\n", input2);
    if( input1 != NULL )    // 此时input1指向的是input2的空间
    {
        strcpy (input1, "hello world"); //此时更改的是input2
    }
    printf("%s\n", input1); // 两个指针所指内容相同
    printf("%s\n", input2);
}
```

指针类型转换的风险

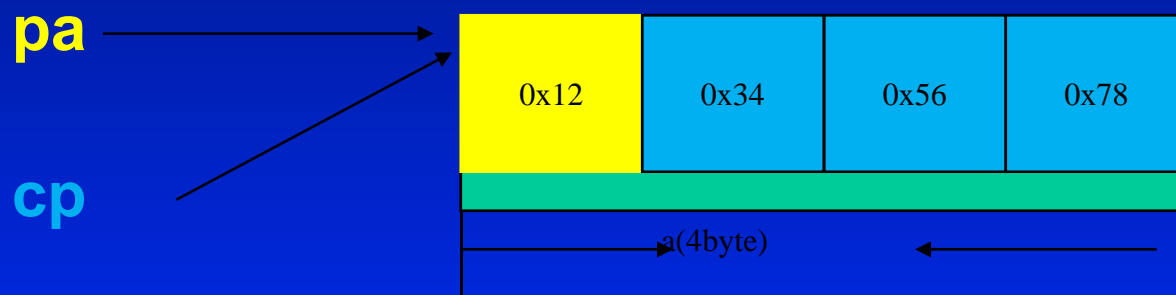
- 大存储类型的指针向小存储类型的指针转换，会带来错误风险

如 `int a = 0x12345678;`

`int *pa = &a;`

`char *pc = (char*)pa;`

`(*pa)` 的数值为 `0x12345678`; `(*pc)` 的数值为 `0x12`;



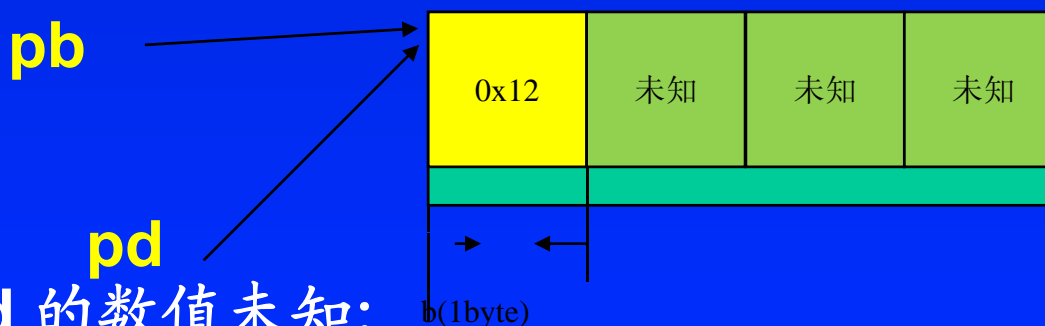
- 小存储类型的指针向大存储类型的指针转换，会带来极大的错误风险

如 `char b = 0x12;`

`char *pb = &b;`

`int *pd = (int*)pb;`

`*pb` 的数值为 `0x12`, `*pd` 的数值未知;



什么是数组名

数组名是一个地址，而且还是一个不可修改的常地址。
完整的说，数组名就是一个地址常量。

数组名这个符号代表了数组内存的首地址。

注意：不是这个符号的值是首地址，是这个符号本身就代表一个地址。

由于数组名是一个符号常量，因此它只能是个右值，
而指针作为变量是个左值，
数组名永远都不会是一个指针变量。

C语言中对于多维数组的实现

C语言只能用数组的数组当作多维数组。

数组的数组与多维数组的主要区别，就在于数组的数组

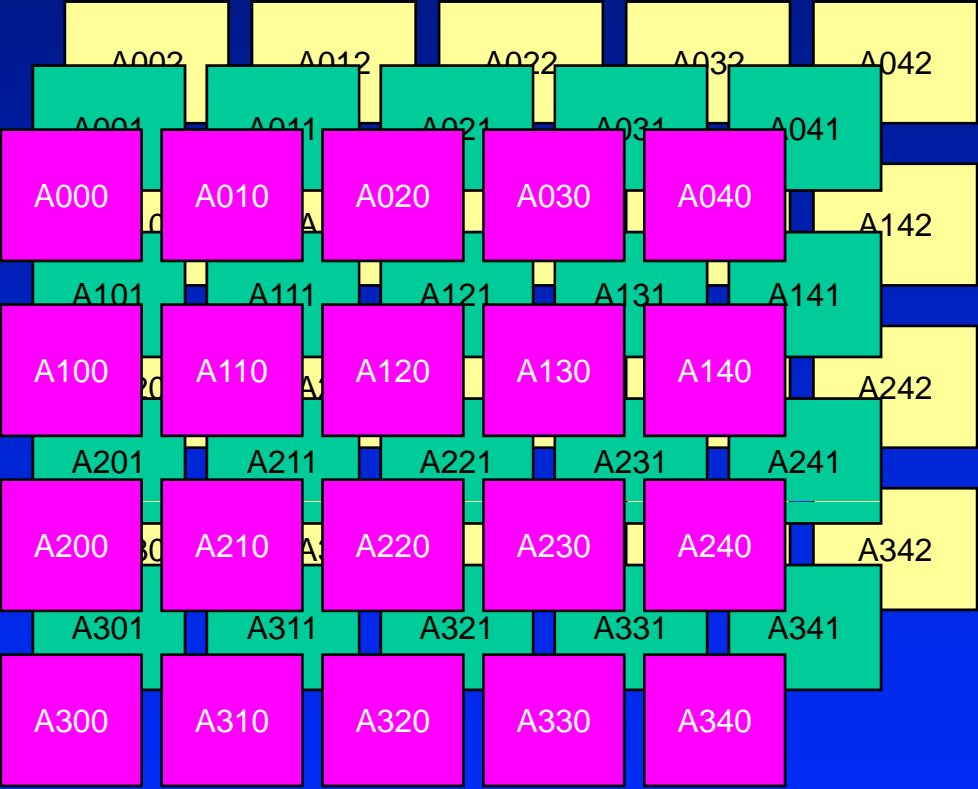
各维之间的内在关系是一种鲜明的层级关系。

上一维把下一维看作下一级数组，也就是数组嵌套。

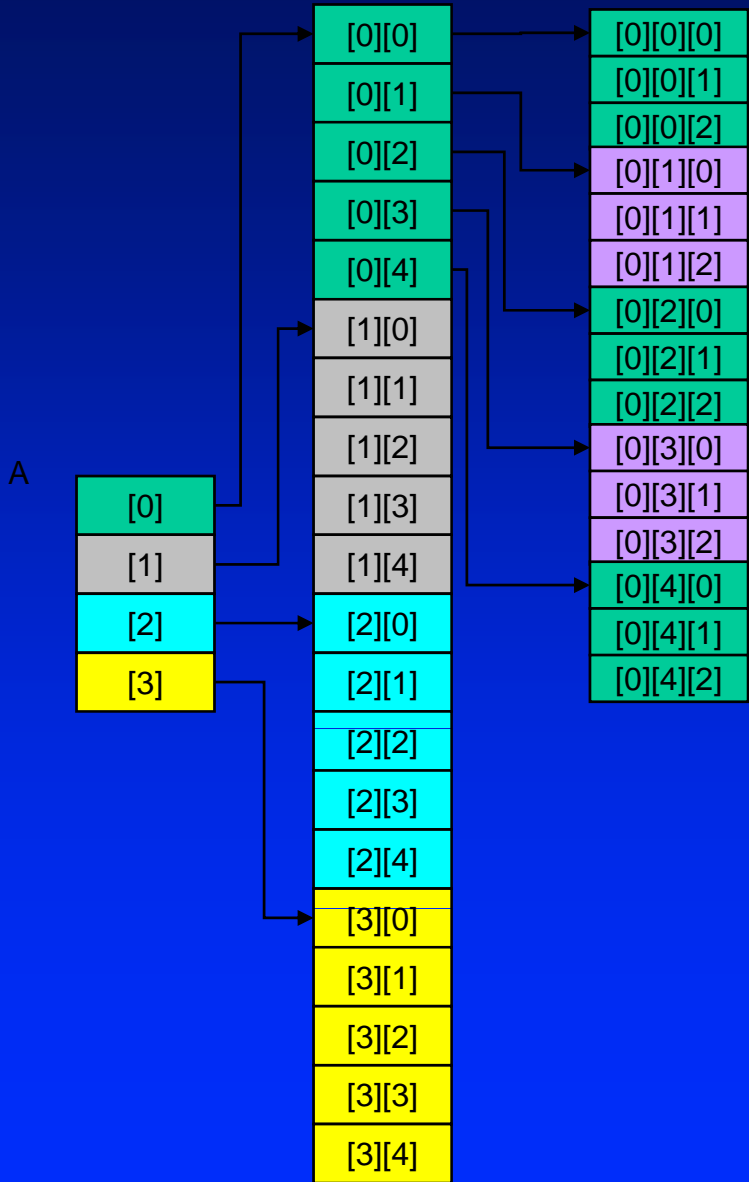
类型是层次间联系的纽带。

数组引用时需要层层解析，直到最后一维。

A[4][5][3] 多维数组外观



实际存储的格式



指针与数组

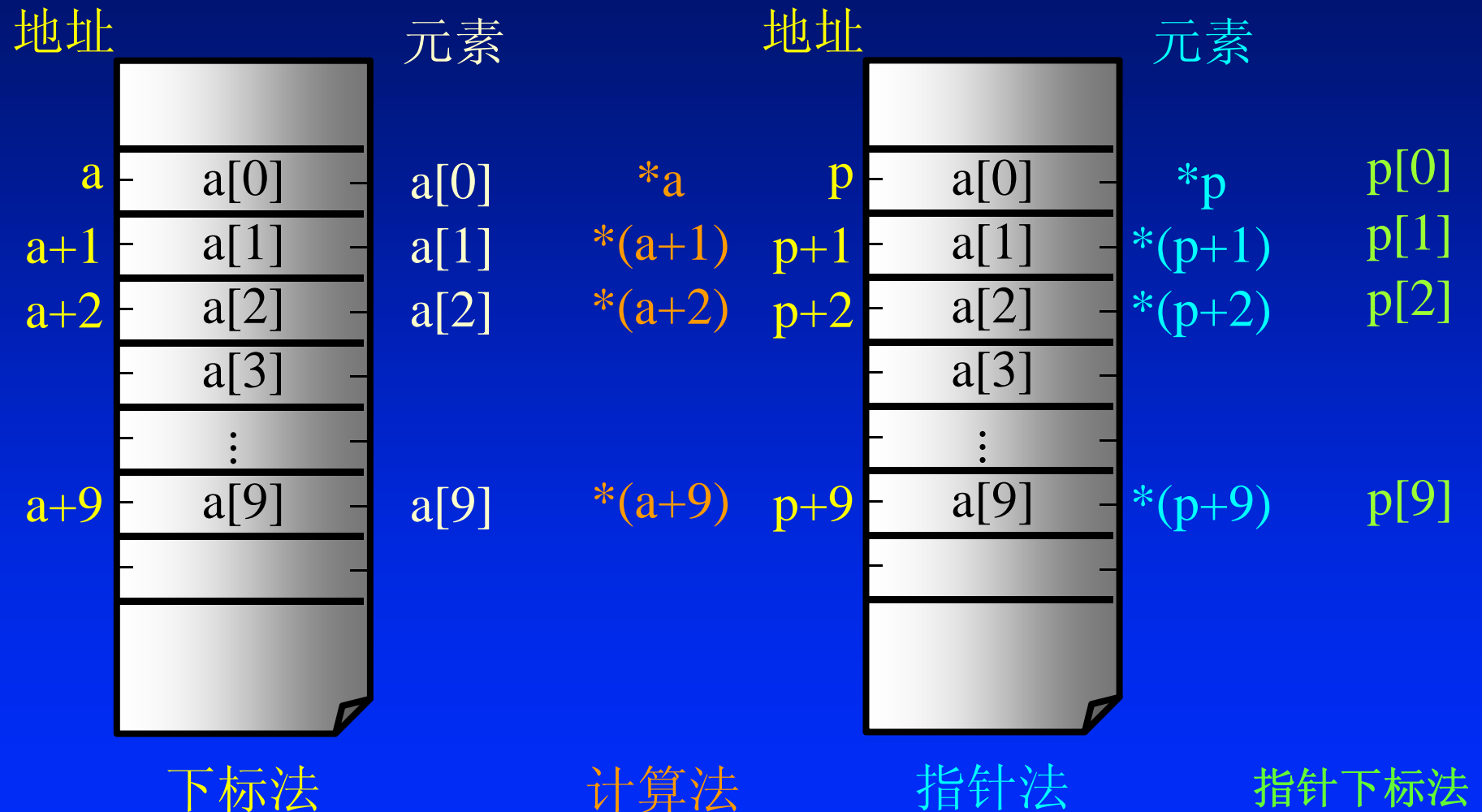
永恒的公式： 间址（首址 + 偏移量）

“指针即含有一维数组！”

与一维数组的关系：指向一维数组的指针。

与N维数组的关系：指向N-1维数组的指针。

访问数组元素



$a[i] \Leftrightarrow *(a+i) \Leftrightarrow *(p+i) \Leftrightarrow p[i]$

三种方法的统一:

	首址	偏移量	运算
下标法	a	i	$a[i]$
计算法	a	i	$*(a+i)$
指针法	p	已用 $p++$ 移到当前位置	$*p$

三种方法的本质都是: 间址 (变址)

无论是下标法 ($a[i]$) 还是计算法 ($*(a+i)$) 还是指针法 ($*(p+i)$), 尽管表现形式不同, 可本质都是:

$*(\text{首址} + \text{偏移量})$

与指针相关的运算符

	名称	定义符	运算符
*	指针运算符	定义了一个指针 <code>int *p = &i;</code>	间址运算符(求值) <code>*p = 30;</code>
&	取址运算符	定义了一个引用 <code>int &r = i;</code>	取址运算符(求址) <code>&i</code>
[]	下标运算符	定义了一个数组 <code>int a[10];</code>	下标运算符(求元素) <code>a[4] = -1;</code>

下标运算符[] 则是 $\ast(\text{首址} + \text{偏移量})$ 。其中含有多重运算：先求类型尺寸，再乘，再加，变址后，再间址。括号提高了运算优先级。

对二维数组元素的访问:

因**a**是二维数组名，代表二维数组的首地址，不能用***a**来获得**a[0][0]**的值，***a**只相当于**a[0]**;

正确的访问形式是:

```
int i, a[ 3 ][ 4 ] = { { 1,2 }, { 3,4,5 }, { 6,7,8,9 } };
```

```
int (*p)[4] = a;
```

```
i=a[0][1];
```

下标法

或

```
i=*(*(a+0)+1);
```

计算法

或

```
i=*(*(p+0)+1);
```

指针法

或

```
i=*(a[0]+1);
```

下标计算法

或

```
i=*(a+0)[1];
```

计算下标法

或

```
i=*(p+0)[1];
```

指针计算下标法

或

```
i=*(p[0]+1);
```

指针下标计算法



C语言规定：数组名代表了数组的地址，所以 $a[i]$ 是第 i 行那个一维数组的地址，它指向该行的第0列元素。 $a[i]$ 可视为“一个一维数组的列指针”。

同理， a 可视为“一个以一维数组为大元素的一维数组的指针”。

以此作为递归单元，即可推导出 N 维数组。

对于： `int A[4][5][3];`

定位一个具体的值, 如 `A[3][4][1]`，只要

`*(*(*(A+3)+4)+1)`

C语言的数组实现并非真正的多维数组，而是数组嵌套，访问某个元素的时候，需要逐层向下解析。

第一维元素 `A[0]` 是 `A[0]` 所代表的那个数组的首地址，但是这个表达式在 C 里面有特殊意义。特殊之处在于它所代表的东西与一般的地址不同。而且类型也不是一般的地址类型，叫数组类型。

若有 `int i = 10;`

`int *p = &i;`

`&*p` 意味着什么?

`*&i` 又意味着什么?

p
i

当遇到`[][]`运算符的时候，编译器只是简单地把它转换为类似`*(*(A+i)+j)`这样的等价表达式。

例如 `A[B]` 等价于 `*(A+B)`。

对于例如 `&A[B]` 的操作，编译器将直接将其优化成 `A+B`。

数组类型跟一般地址类型的区别

- 最主要的区别就是长度不同
- 一般的地址长度为4（不同的系统可能会不一样）
- 数组类型的长度代表的数组的长度。

如在 `int A[5][3];` 中 `sizeof(A[0]) == 3*sizeof(int)`

数组类型在数组的定义与引用中具有非常重要的作用，它可以用来识别一个标识符或表达式是否真正的数组。一个真正数组的数组名，是一个具有数组类型的地址常量，它的长度，是整个数组的长度，并非一个一般地址的长度，如果一个标识符不具备数组类型，它就不是一个真正的数组。

`&a[0][0][0]`仅仅是一个地址，它的意义，仅仅表示元素 `a[0][0][0]` 的地址。 `sizeof(&a[0][0][0])` 的结果只是4。

不少人把它说成是数组 `a` 的首地址，错误！这是对数组首地址概念的滥用。真正能代表数组 `a` 的数组首地址只有 `a` 本身，数组首地址是具有数组类型的地址， `sizeof(a)` 结果是 `i*j*k*sizeof(int)`，而不是4。只不过由于 `a[0][0][0]` 位置特殊，是数组 `a` 的第一个元素，所以它们的地址值才相同。

虽然指针和数组都表示地址，有时可以混用、相互替换。
但它们是有区别的。

指针指向的是内存块，可以改变；而数组名是某块固定内存的地址，不能改变。

如：char a[10], *p;

p = a; //OK

char a[10], b[10];

a = b; // error

通过数组名可以求得数据块的大小，通过指针则不能

```
int a[10];
```

```
int *p;
```

```
p = a;
```

```
printf( "%d", sizeof( a ));    /* output is 40 */
```

```
printf( "%d", sizeof( p ));    /* output is 4 */
```

对于指针和数组的区别还表现在使用scanf和printf上:

```
void main()
```

```
{
```

```
    char name[14], *p=name;
```

```
    scanf("%s", &name);
```

```
    scanf("%s", name);
```

对于数组名可以带&也可以不带

```
    scanf("%s", p);
```

```
    scanf("%s", &p);
```

但对于指针则一定不能带&

```
    printf("%s\n", &name);
```

```
    printf("%s\n", name);
```

对于数组名可以带&也可以不带

```
    printf("%s\n", p);
```

```
    printf("%s\n", &p);
```

但对于指针则一定不能带&

```
}
```

当数组作为函数的参数进行传递时，该数组自动蜕化为同类型的指针。

因为当初**ANSI**委员会制定标准时，从**C**程序的执行效率出发，不主张参数传递时复制整个数组，而是传递数组的首地址，由被调函数根据这个首址处理数组中的内容。那么谁能承担这种“转换”呢？主体必须具有地址数据类型，同时应该是一个变量，满足这两个条件的，当然非指针莫属。

指向多维数组的指针

通常，对于 `int a[8][9]` 这个二维数组，我们可以定义一个指向它的指针：

`int (*p)[9];` 这叫“指向二维数组指针变量”。

它的一般说明形式为：

类型说明符 (*指针变量名)[长度]

其中“类型说明符”为所指数组的数据类型。“*”表

示其后的变量是指针类型。“长度”表示二维数组分解为多个一维数组时，一维数组的长度，也就是二维数组的列数。

思考： `int (*p)[9]` 和 `int *p[9]` 的区别？

```
void main()
{
    static int a[3][4]={0,1,2,3},{4,5,6,7},{8,9,10,11}};
    int(*p)[4];
    int i,j;
    p=a;
    for(i=0;i<3;i++)
        for(j=0;j<4;j++)
            printf("%2d ",*(*(p+i)+j));
}
```

类似的,对于 一个指向**N**维数组的指针可以这样定义:

类型说明符 (*指针变量名)[长度][长度].....[长度]
int (*p)[x2][x3].....[xn];

一个例子: **int (* ptr)[3];**
 ptr=new int[2][3];
 int (* ptr2)[3][5];
 ptr2=new int[4][3][5];

再谈指针与数组

读这段代码，给出输出结果，并解释为什么？

```
#include
```

```
void m
```

```
{
```

```
int a[5]={1,2,3,4,5};
```

```
int *ptr=(int *)(&a+1); //先取地址再加1
```

```
//printf("%p,%p\n",&a+1,ptr); // 将地
```

```
printf("%d,%d\n",*(a+1),*(ptr-1));
```

```
}
```

可见，“类型”起着关键作用！

请注意**&a**的含义！
&a的类型是**int(*) [5]**型，加1已经在界外，强转是恢复其普通地址的身份。

输出：2,5
Why?

(a+1)**就是**a[1]**，(ptr-1)**就是**a[4]**，执行结果是2,5。

解释:

对于 `int *ptr=(int *)(&a+1);` 系统不认为 `&a+1` 是数组首地址+1。对数组名求址 (`&a`)，会导致数组维数升级，变成了数组指针，其类型为 `int (*)[5]`。而这样的指针加1，则要根据指针类型加上一个 `a` 数组的偏移，是偏移了整个数组的大小（本例是5个 `int`），所以要加 `5*sizeof(int)`。于是 `ptr` 实际是 `&(a[5])`，也就是 `a+5` 了，`ptr` 实际指向 `a[5]`。

但，`pri` 与 `(&a+1)` 类型是不相同的，`pri-1` 只会减去一个整型位置，即 `sizeof(int*)`。

尽管 `a`、`&a` 的地址是同一个，但含意是不一样的。`a` 是数组首地址，也就是 `a[0]` 的地址，`&a` 是对象（数组整体）首地址。`a+1` 是数组下一元素的地址，即 `a[1]`，而 `&a+1` 是下一个数组的地址，即 `a[5]`。

众所周知，C语言是没有字符串变量的，因而，C89规定，字符串常量就是一个字符数组。因此，尽管字符串常量的外部表现形式跟数组完全不同，但它的确是一个真正的数组。

字符串常量本身就是这个数组的首地址，并且具有数组类型，对一个字符串常量进行sizeof运算，例如sizeof("abcdefghi")，结果是10，而不是4。

指针与数组的区别:

```
char str[] = "hello";
```

```
char *p = str;
```

```
int n = 10;
```

在32位计算机上, 计算sizeof的值:

```
sizeof (str) = ?
```

```
sizeof (p) = ?
```

```
sizeof (n) = ?
```

```
void Fun(char str[100])
```

```
{
```

```
    sizeof (str) = ?
```

```
}
```

字符串数组与一般数组的区别

字符串常量存放在静态存储区，而一般数组（非**static**）则是在栈中静态分配的。

由于字符串常量是数组首地址，因此可以数组引用的形式使用它，例如：

```
printf("%s", &"abcdefghi"[4]);
```

这将打印出字符串efghi。还可以这样：

```
printf("%s", "abcdefghi"+4);
```

同样打印出字符串efghi。实际上，&“abcdefghi”[4]等价于&*(“abcdefghi”+4)，去掉&*后，就是“abcdefghi”+4了

字符串的问题

【问题1】请问以下(1)和 (2)有区别吗?

(1) `char str[] = "HelloWorld";`

`char *p = str;`

(2) `char *p = "HelloWorld";`

【问题2】请问b和c的值各是多少?

`char a[] = "Hello";`

`int b, c;`

`b = sizeof(a);`

`c = strlen(a);`

对。可以通过数组名或指针去修改。

错。因为指针指向的是常量区，而p却是个指向变量的指针，那意味着可改所指的数据，故应视为错误:缺**const**修饰。

【问题3】 回答以下问题:

```
char str[ ] = "HelloWorld";
```

```
char *p = "HelloChina";
```

(1) 可以这样做吗? `p = str ;`

(2) 这样做结果会怎样?

(3) 串“HelloChina” 会消失吗?

(4) 这样可以吗? `str = p;`

(5) 可以这样做吗? 为什么?

```
str[ 1 ] = 'u';
```

```
p [ 1 ] = 'u' ;
```

[例] 有若干计算机图书，请按字母顺序，从小到大输出书名。解题要求：使用排序函数完成排序，在主函数中进行输入输出。

/*案例代码文件名：AL9_12.C*/

/*程序功能：指针数组应用示例*/

```

/* sort()函数: 对字符指针数组进行排序 */
/*形参: name——字符指针数组, count——元素个数*/
/*返回值: 无 */
/*****/
void sort(char *name[], int count)
{
    char *temp_p;
    int i,j,min;
    /*使用选择法排序*/
    for(i=0; i<count-1; i++) /*外循环: 控制选择次数*/
    { min=i; /*预置本次最小串的位置*/
        for(j=i+1; j<count; j++) /*内循环: 选出本次的最小串*/
            if(strcmp(name[min],name[j])>0) /*若存在更小的串*/
                min=j; /*则保存之*/
        if(min!=i) /*存在更小的串, 交换位置*/
            temp_p=name[i],name[i]=name[min],name[min]=temp_p;
    }
}

```

```
main() /*主函数main()*/  
{  
    char *name[5]=  
        {"BASIC","FORTRAN","PASCAL","C","FoxBASE"};  
    int i=0;  
    sort(name,5); /*使用数组名作实参，调用排序函数sort()*/  
    /*输出排序结果*/  
    for(; i<5; i++)  
        printf("%s\n",name[i]);  
}
```

程序运行结果:

BASIC

C

FORTRAN

FoxBASE

PASCAL

程序说明:

(1) 实参对形参的值传递:

sort(name , 5);



void sort(char *name[], int count);

(2) 字符串的比较只能使用**strcmp()**函数。

形参字符指针数组**name**的每个元素，都是一个指向字符串的指针，所以有**strcmp(name[min],name[j])**。

【例】“锯齿数组” 分配内存和释放内存的过程
样例。

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
void main()
```

```
{
```

```
int a1 = 5;
```

```
int a2[ ] = {4,2,3,2,4};
```

```
int i,j;
```

```
int * * p_array;
```

```
p_array = (int**)calloc(a1,sizeof(int *));
```

```
for(i=0; i<a1; i++)
```

```
    p_array[i] = (int*)calloc(a2[i],sizeof(int));
```

```
for(i=0; i<a1; i++)  
    for(j=0; j<a2[i]; j++)  
        p_array[i][j] = 1+2*i*j;  
for(i=0; i<a1; i++)  
{  
    for(j=0; j<a2[i]; j++)  
        printf("%d ",p_array[i][j]);  
    printf("\n");  
}  
for(i=0; i<a1; i++)  
    free(p_array[i]);  
free(p_array);  
}
```

为何这样释放？

判断下列 程序 的运行结果：

```
#include <stdio.h>
#include <string.h>
void main()
{  int a[]={10,15,4,25,3,-4};
    int * p=&a[2];
    printf("%d\n" , *(p+1)) ;
    printf("%d\n" , p[-1]) ;
    printf("%d\n" , p-a) ;
    printf("%d\n" , a[*p++]) ;
    printf("%d\n" , *(a+a[2])) ;
}
```

该程序显示什么？

```
#include <stdio.h>
```

```
#include <string.h>
```

```
void main()
```

```
{
```

```
    char s[]="computer", * p = s;
```

```
    printf("%s\n" , s) ;
```

```
    printf("\n%c\n" , *p++);
```

```
    printf("%c\n" , *(p++));
```

```
    printf("%c\n" , (*p)++);
```

```
    printf("%s\n" , s) ;
```

```
printf("\n%c\n" , *++p) ;
```

```
printf("%c\n" , *(++p)) ;
```

```
printf("%c\n" , ++*p) ;
```

```
printf("%c\n" , ++(*p)) ;
```

```
printf("\n%s\n" , s) ;
```

```
printf("\n%s\n" , p) ;
```

```
}
```

又: `strcmp(s , ?) == 0`

该程序显示什么？

```
#include <stdio.h>
#include <string.h>
```

```
void main()
```

```
{
```

```
    char * msg = "Hello world !";
```

```
    printf(" %s \n", msg ) ;
```

```
    strcpy(msg , "Hi !");    //试图修改常量区！
```

```
    printf(" %s \n", msg ) ;
```

```
}
```

该程序显示什么？

```
void main()
```

```
{
```

```
    void * buf = malloc(0);
```

```
    if ( buf == null )
```

```
        printf(" is null \n" ) ;
```

```
    else
```

```
        printf(" is not null \n" ) ;
```

```
}
```


指针与常(**const**)

情况一：指针指向常量**const int i**

```
const int i=40;
```

```
int *pi=&i; //这样可以吗？不行，编译错。
```

但VC++只给出了警告。

const int 类型的**i**的地址是不能赋给指向**int** 类型地址的指针**pi**的。否则**pi**岂不是能修改**i**的值了吗！

```
pi=(int* ) &i; //这样可以吗？强制类型转换可是C语言所支持的。
```

VC++下编译通过，但是仍不能通过***pi=80**来修改**i**的值。试试！看看具体的结果。

正确的写法是：**const int * pi = &i ;**

情况二：常指针指向变量

```
int i=40;
```

```
int * const pi = &i;
```

此时指针是常量，它忠心耿耿的指着i，决不见异思迁。但是通过pi却可以修改i的值，因为i是变量。

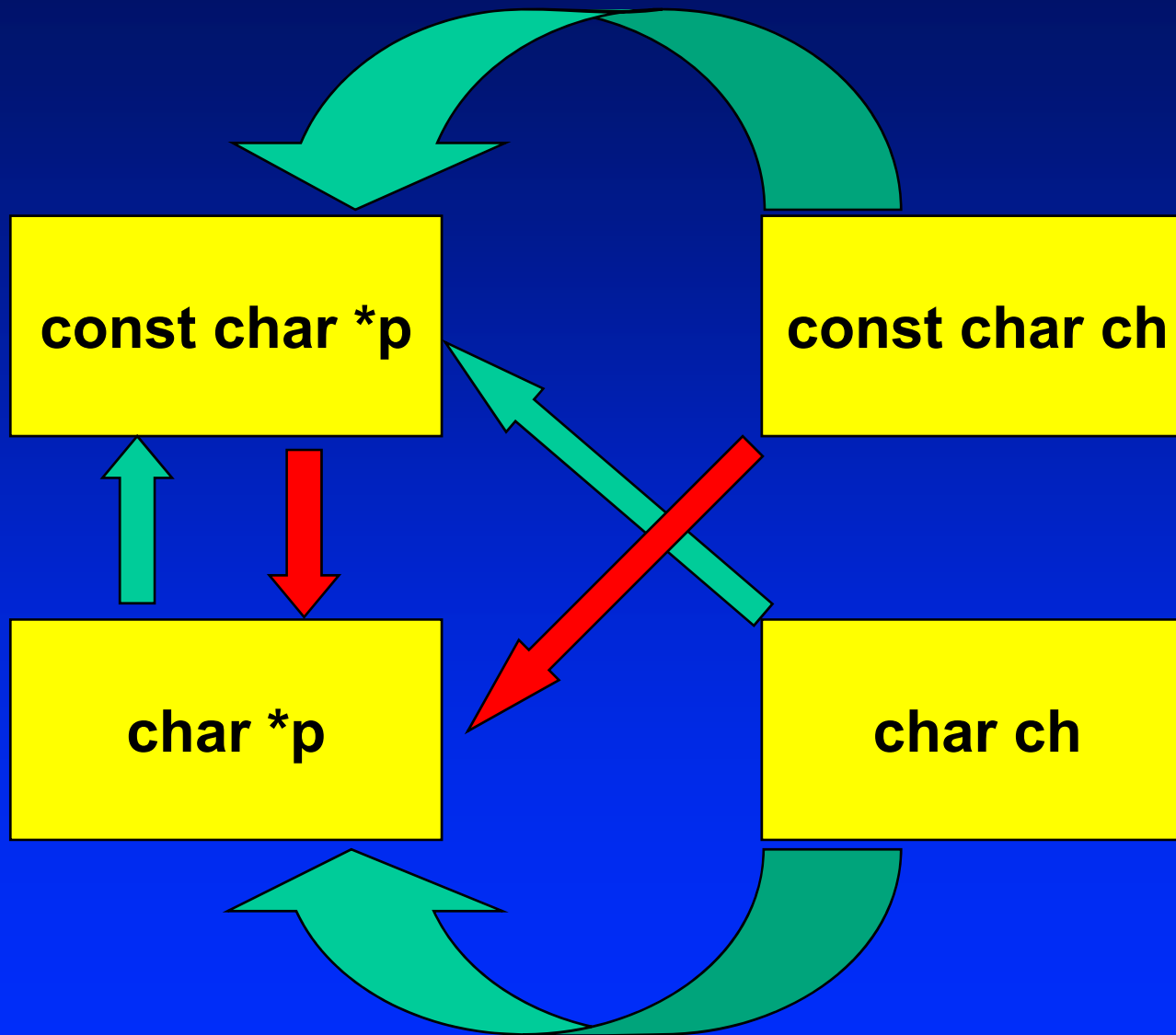
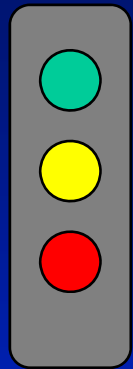
情况三：指向常量的常指针

```
const int i;
```

```
const int * const pi=&i;
```

你能想像pi能够作什么操作吗？pi值不能改，也不能通过pi修改i的值。因为不管是*pi还是pi都是const的。

const与non-const的赋值问题



“ const * = non-const * ”

const 指针可以得到变量的地址。

```
const char *p = NULL;
```

```
char ch = 'a';
```

```
p = &ch;    //正确!
```

非**const**指针不可以用**const**对象的地址赋值:

```
char *p = NULL;
```

```
const char ch = 'a';
```

```
p = &ch;    //错误!
```

练习:

```
void main()
```

```
{
```

```
    char *p1 = NULL;
```

```
    char **p = &p1;
```

```
    const char **q = NULL;
```

```
//以下语句都当作独立的语句处理。
```

```
    q = p;        //right
```

```
    p = q;        //error
```

```
    *p = *q;      //error
```

```
    *q = *p;      //right
```

```
}
```

解释:

q = p;正确!

q是指向**const char ***类型的指针。

而**p**是指向**char ***类型的指针。

“**const = non-const**”



p = q; 错误!

因为这样**p**经过两次间址就可以修改**const**对象的值，编译器不允许这样。

“**non-const = const**”



指针与函数

这叫返回类型

这叫参数类型

指针出现在函数的三个地方:

返回类型 函数名 (形参表);
(1) (2) (3)

合起来叫函数类型

可以依此证实:

```
typedef float (*pf) (int, double);
```

pf 是个类型名

指针与函数参数

函数的形参表中可以含有指针。有多种形式:

1. 指针作形参, 如**void f(int *p);** 作用是对作为变量的实参传址, 如 **f (&a);**
2. 指向指针的指针作形参, 如**void f(int **pp);** 作用是对作为指针的实参传址, 如 **f (&p);**
3. 指针的引用作形参, 如**void f(int *&rp);** 作用是对作为指针的实参传址, 如 **f (p);**

指针变量，既可以作为函数的形参，也可以作函数的实参。

函数的形参即使写成数组形式，其实质也是指针：

void fun(int * ptr)和void fun(int ptr[])
是等价的。甚至可以写成：

void fun(int *)和void fun(int [])

指针变量作实参时，与普通变量一样，也是“值传递”，即将指针变量的值（一个地址）传递给被调用函数的形参（必须是一个指针变量）。

请注意：被调用函数可以改变实参指针变量所指向的变量的值。也可以改变形参的值，但对于实参指针变量没有影响——改了也白改。

```
void Allocate( char * & p , int size)
```

```
{
```

```
    p = (char*) malloc(size)
```

```
}
```

此代码作为 char * p 行吗?

改为 char ** p 呢?

```
void Test (void )
```

```
{
```

```
    char *str = NULL;
```

```
    Allocate ( str , 100 );
```

```
    strcpy ( str , "Hello World!" );
```

```
    printf ( str );    // 可以这样用
```

```
    free ( str );
```

```
}
```

用字符指针变量作形参:

```
void copy_string(char *from, char *to) // 指针作形参
```

```
{  
    for (; *from != '\0'; *to++ = *from++)  
        *to++;  
    *to = '\0';  
    from++;  
}  
main()  
{  
    char a[] = "I am a teacher";  
    char b[] = "You are a student";  
    printf("String a = %s\n", a);  
    copy_string(a, b);  
    printf("String b = %s\n", b);  
}
```

否? 应视为:

***to = *from;**

to++;

from++;

作左值的不是后++,

而是指针自身, 这

完全不同于:

i++ = j++;

此处的***to++=**属“后++不能作左值”的情况吗?

注意: a和b可以换成指针吗? 为什么?

(*to++=*from++)!= '\0';

主函数main()的形参

在以往的程序中，主函数main()都使用无参形式。实际上，主函数main()也是可以指定形参的。

主函数main()的有参形式：

```
main(int argc, char *argv[])  
{    ...    }
```

实参的来源：

运行带形参的主函数，必须在操作系统状态下，输入主函数所在的可执行文件名，以及所需的实参，然后回车即可。

命令行的一般格式为：

可执行文件名 实参[实参2.....]

本案例程序的用法：lock +|- <被处理的文件名>←┘

[案例9.13] 用同一程序实现文件的加密和解密。

约定：程序的可执行文件名为**lock.exe**；

其用法为：**lock +|-** <被处理的文件名>

其中 “+”为加密，“-”为解密。

*/*案例代码文件名：AL9_13.C*/*

*/*程序功能：带参主函数的应用示例*/*

```
main(int argc, char *argv[])
{ char c;
  if (argc != 3) printf("参数个数不对! \n");
  else
  { c=*argv[1]; /*截取第二个实参字符串的第一个字符*/
    switch(c)
    { case '+':      /*执行加密*/
      { /*加密程序段*/
        printf("执行加密程序段。 \n");
      } break;
      case '-':      /*执行解密*/
      { /*解密程序段*/
        printf("执行解密程序段。 \n");
      } break;
      default: printf("第二个参数错误! \n");
    }
  }
}
```

形参说明

(1) 形参**argc**是命令行中参数的个数（可执行文件名本身也算一个）。

在本案例中，形参**argc**的值为3 (**lock**、**+|-**、文件名)。

(2) 形参**argv**是一个字符指针数组，即形参**argv**首先是一个数组（元素个数为形参**argc**的值），其元素值都是指向实参字符串的指针。

在本例中，元素**argv[0]**指向第1个实参字符串“**lock**”，元素**argv[1]**指向第2个实参字符串“**+|-**”，元素**argv[2]**指向第3个实参字符串“被处理的文件名”。

函数以及函数名

函数的定义:

```
functiontype functionname( argument list)
{
    .....
    function body
    .....
}
```

函数名:

函数名是个常量(不能修改)。利用函数名可以访问(调用)函数;

那么我们可以定义函数指针, 利用函数指针来完成对函数调用。这个指针应该和所指的函数具有相同的类型。

指向函数的指针

一个通常的函数调用的例子（没使用指针）：

```
void MyFun(int x); //此处的声明也可写成:
```

```
void MyFun( int );
```

```
void main(int argc, char* argv[])
```

```
{
```

```
    MyFun(10); //这里是调用MyFun(10);函数
```

```
}
```

```
void MyFun(int x) //这里定义一个MyFun函数
```

```
{
```

```
    printf(“%d\n”,x);
```

```
}
```

从功能上或者说从数学意义上理解函数，**MyFun**函数名代表的是一个功能（或是说一段代码）。

函数指针变量的声明

如同某一数据变量的内存地址可以存储在相应的指针变量中一样，函数的首地址也可以存储在某个指针变量里。这样，就可以通过这个指针变量来调用所指向的函数了。

在C系列语言中，任何一个变量，总是要先声明，之后才能使用的。那么，函数指针变量也应该先声明。以上面的例子为例，来声明一个可以指向MyFun函数的函数指针变量FunP:

void (*FunP)(int x); 也可写成**void (*FunP)(int);**
函数指针变量的声明格式，如同函数MyFun的声明一样，只不过把MyFun改成(*FunP)而已。这样就有了一个能指向MyFun函数的指针了。当然，这个指针变量也可以指向其它所有具有相同函数格式的函数。

通过函数指针变量调用函数

有了FunP指针变量后，就可以对它赋值，指向MyFun，然后通过FunP来调用MyFun函数了。

```
void MyFun(int x);
```

```
void (*FunP)(int ); //也可声明成void(*FunP)(int x),  
但习惯上一般不这样。
```

```
void main(int argc, char* argv[])
```

```
{
```

```
    MyFun(10); //这是直接调用MyFun函数
```

```
    FunP=MyFun; //对函数指针赋值
```

```
    (*FunP)(20); //通过函数指针FunP来调用MyFun函数
```

```
    FunP(20); //也可以写成这样
```

```
}
```

```
void MyFun (int x) //定义了MyFun函数
{
    printf(“%d\n”,x);
}
```

MyFun与FunP的类型是相吻合的。函数MyFun好像是一个void (int)类型的数组名（常量），而*FunP则是一个void (int)类型的指针变量。

就如同：

```
int A[10],*pi;
pi = A; //与FunP=MyFun比较。
你的感觉呢？
```

继续看以下几种情况：（这些可都是可以正确运行的代码！）

```
void main(int argc, char* argv[])
```

```
{
```

```
    MyFun(10);    //调用MyFun(10);函数
```

```
    FunP=MyFun; //将函数地址赋给指针变量
```

```
    (*MyFun)(10); //函数名MyFun也可以有这样的  
调用格式，就如同用(*FunP)(20); 来调用MyFun函数
```

```
}
```

真的是可以这样的！

依据以往的知识和经验来推理本篇的“新发现”，必定会推断出以下的结论：

1. **MyFun**的函数名与**FunP**函数指针都是一样的，即都是函数指针。**MyFun**函数名是一个函数指针常量，而**FunP**是一个函数数指针变量，这是它们的关系。

2. 但函数名调用如果都得如**(*MyFun)(10)**；这样，那书写与读起来都是不方便和不习惯的。所以C语言的设计者们才会设计成可允许**MyFun(10)**；这种形式地调用（这样方便多了并与数学中的函数形式一样）。

3. 为统一起见，**FunP**函数指针变量也可以**FunP(10)**的形式来调用。

4. 赋值时，即可**FunP=&MyFun**形式，也可**FunP=MyFun**。

定义函数的指针类型:

就像自定义数据类型一样，可以先定义一个函数指针类型，然后再用这个类型来声明函数指针变量。

下面先给出一个自定义数据类型的例子：

```
typedef int* PINT; //为int* 类型定义了一个PINT的  
                  别名
```

```
void main()
```

```
{
```

```
    int x;
```

```
    PINT px=&x; //与int * px=&x;是等价的。PINT类型
```

其实就是int * 类型。

```
    *px=10;      //px就是int*类型的变量。
```

```
}
```

下面来看一下函数指针类型的定义及使用：（请与上对照！）

```
void MyFun(int x); //此处的声明也可写成：  
void MyFun( int );
```

```
typedef void (*FunType)(int ); //这样只是定义一个  
函数指针类型
```

```
FunType FunP; //然后就可以用FunType类型来声明  
FunP变量
```

```
void main(int argc, char* argv[])
{
    FunType FunP;    //声明了函数指针变量
    MyFun(10);
    FunP=&MyFun;
    (*FunP)(20);
}
```

```
void MyFun(int x)
{
    printf("%d\n",x);
}
```

解析:

首先，在`void (*FunType)(int);`前加了一个`typedef`。这只是定义一个名为`FunType`的函数指针类型，而不是`FunType`变量。

然后使用`FunType FunP;`这句就如`PINT px;`一样地声明了一个`FunP`变量。

其它相同。整个程序完成了相同的事。

这样做法的好处是：

有了`FunType`类型后，我们就可以同样地、很方便地用`FunType`类型来声明多个同类型的函数指针变量了。如下面的代码：

```
FunType FunP2, FunP3;
```

指向函数的指针的使用

```
int fun1(char*,int);    char arr[ ] = "china!";
int fun2(char*,int);
void fun3(double *,int);
int(*pfun)(char*,int); // pfun1是指向函数的指针
pfun = fun1;           //给函数指针赋值
int a = (*pfun)("abcdefg", 7); //用指向函数的指针进行
函数调用
pfun = fun2;           //指向函数的指针可以被修改
a = (*pfun)( arr, 6); // 尽管类型不完全相符，亦可。
pfun = fun3;           //错，类型不相符
pfun = fun2 ( ) ;      // 错。    Why?
```

观察下面的函数原型:

```
void show ( int (*FP)(char *) , char * );
```

其中: 第一形参是指向函数的指针, 第二形参是指向数据的指针变量。于是可以这样来调用:

```
show ( fun1, arr ); 或 show ( pfun, arr );
```

这是一个以函数作为参数的 例题:

各种指针

请说明各指针的含义:

`int *ptr;`

`char *ptr;`

`int **ptr;`

`int *ptr[3];`

一维指针数组

`int (*ptr)[3]`

指向二维数组的指针

`int *ptr [3][4];`

二维指针数组

`int *(*ptr)[4];`

指向二维指针数组的指针

`void* (*ptr)(void*);`

指向函数的指针

`int * pMove();`

返回指针的函数

`int (*p[3]) (int);`

函数指针数组,函数返回int型数据

`int (*p) [3] [4];`

指向三维数组的指针

`int (*p[3]) [4];`

指针数组的每个元素都是指向二维数组的指针

观察总结1:

“指向一个元素的指针”是指向一维数组的指针:

```
int *ptr;
```

```
int a[N];
```

```
ptr = a;
```

“指向一行元素的指针”是指向二维数组的指针:

```
int (*ptr)[3];
```

```
int a[M][N];
```

```
ptr = a;
```

观察总结2:

int *ptr[3];



int *ptr [3];

int (*ptr)[3];



int [3] (*ptr);



int *(*ptr)[4];



(int *) (*ptr)[4];

int (*ptr[3])[4];



int (*ptr[3])[4];

那么大家考虑:

int (* (*func)(int *p))[5];

char (*(* fun ())[4]) (int *p);

int *(*(*func)(int *(*)[10]))[10];

复杂指针解析的方法——分解法

首先从未定义的标识符起往右看，当遇到圆括号时，就掉转方向往左看。解析完圆括号里面所有的东西，就整个给它一个命名。将命名与外面的东西再重复这个过程直到将整个声明解析完毕。**外面的东西其实是类型。**

■ 第一个例子

```
int (*func)(int *p);
```

首先找到未定义的标识符，就是**func**，它的外面有一对圆括号，而且左边是个*号，这说明**func是个指针**，然后跳出这个圆括号，并给它一个命名，比如**F**。再看右边，也是一个圆括号，这说明**F是个函数**，而**func是个指向这类函数的指针**，这类函数具有**int***类型的形参，返回值类型是**int**。

■ 第二个例子

```
int (*func)(int *p, int (*f)(int*));
```

func被一对括号包含，且左边有个*号，说明**func**是个指针，给它一个命名，比如**F**。其右边也有个括号，那么**func**是个指向函数的指针，这类函数具有**int ***和**int (*)(int*)**形参，返回值为**int**类型。再来看形参**int (*f)(int*)**，类似前面的解释，**f**也是一个函数指针，指向的函数具有**int***类型的形参，返回值为**int**

■ 第三个例子

```
int (*func[5])(int *p);
```

func右边是一个[]运算符，说明**func**是具有5个元素的数组，**func**的左边有个*，说明**func**的数组元素是指针，要注意这里的*不是修饰**func**的，而是修饰**func[5]**的，原因是[]运算符优先级比*高，**func**先跟[5]结合。

给它一个命名，比如**F**。看右边，也是一对圆括号，说明**func**数组的元素是函数类型的指针，它所指向的函数具有**int***类型的形参，返回值类型为**int**。

所以**func**是“每个元素都是指向函数的指针的数组名”。

■ 第四个例子

```
int (* (*func)[5] )(int *p);
```

func被圆括号包含，左边又有*，那么**func**是个指针，右边是个[]号，那么说明**func**是一个指向数组的指针，给它一个命名，比如**F**，那么其余部分就是数组的**类型**了。往左看，有个*号，说明这个数组的元素是指针，跳出括号，右边又有一对括号，说明是个形参。这说明数组的元素是指向函数的指针。总结一下，就是：**func**是一个指向二维数组的指针，这个数组的每个元素都是函数指针，这些指针指向具有**int***形参，返回值为**int**类型的函数。

从以上解析可以看出，**类型**是揭示谜底的法宝。

解读:

char (* (* (* q) ()) []) () ;

(* q) () —— k 然后其它的东西都是它的返回类型

(* k) [] —— m 这个数组的类型在下面

char (* m) () 这是数组的类型

总结: q是个指向函数的指针, 该函数无形参, 它的返回类型是指向二维数组的指针, 数组元素的类型又是指向函数的指针, 无形参, 返回值是字符型。

```
char ( * ( * ( * q ) ( ) ) [ 3 ] ) ( ) ;
```



```
char ( * ( * ) [ 3 ] ) ( ) ( * q ) ( ) ;
```



```
char ( * ) ( ) ( * ) [ 3 ] ( * q ) ( ) ;
```



结论: q是个指向函数的指针, 该函数无形参, 它的返回类型是指向二维数组的指针, 数组元素的类型又是指向函数的指针, 无形参, 返回值是字符型。

■ 第五个例子

```
int (* (*func)(int *p) )[5];
```

func是一个函数指针，这类函数具有**int***类型的形参（红的部分）。返回值（黑的部分）是指向数组的指针，该指针所指向的数组的元素是具有**5个int**元素的数组 **int (*)[5]**。

■ 刚才这个例子 `int (* (*func)(int *p))[5];` 的解析:

`int (* (*func)(int *p))[5];`



`int (*)[5] (*func)(int *p) ;`

`int (*)[5] (*)(int *p) func ;`

■ 第六个例子

```
char (*( * fun () )[4]) (int *p)
```

从fun ()看fun是个函数，将其命名为F，得：

```
char (*( * F )[4]) (int *p) ;
```

分析(* F)[4]，F是指向二维数组的指针，命名为P得

```
char (*P) (int *p) ;
```


可见P是个指向函数的指针。

于是，fun是一个函数，返回值是指向二维数组的指针。所指向的数组的元素都是指针，每个指针都可以指向函数。这类函数具有返回char类型、形参是一个且类型是整型指针的架构。

■ 第六个例子

char (*(* fun ()) [4]) (int *p)

char (*(* fun ()) [4]) (int *p);



char (*(*) [4]) (int *p) fun () ;



char (*) (int *p) (*) [4] fun () ;



■ 第七个例子

```
int *(*func)(int (*)(int))[10];
```

func是一个指向函数指针，这类函数用指向二维的指针数组的指针做参数，返回值的也是同类型指针。

此句整体定义了一个指向函数的指针数组。

若定义了**typedef int * (*pa)[10];**后，就可以用

pa(*func)(pa); 来代替该句。

■ 第七个例子

```
int *(*func)(int (*)(int *))[10];
```

```
int *( *func)(int * ) [10];
```



```
int * ( * ) [10] (*func) ( int * ) ;
```

此时回想 `int *(*ptr)[4]` 是什么？



指向二维指针数组的指针

解读

- 1) `int (*(func)[5][6])[7][8];`
- 2) `int (*(func)(int *))[5](int *);`
- 3) `int (*(func[7][8][9])(int *))[5];`

1) **func**是指向三维数组的指针，这类数组的大元素是具有5X6个int元素的二维数组，而指向三维数组的指针又是另一个三维指针数组的元素。

2) **func**是函数指针，这类函数的返回值是指向数组的指针，所指向数组的元素也是函数指针，指向的函数具有int*形参，返回值为int。

3) **func**是个三维指针数组，数组元素是函数指针，这类函数具有int*的形参，返回值是指向数组的指针，所指向的数组的元素是具有5个int元素的数组。

指向指针的指针

```
int i = 20;
```

```
int *pi = &i;
```

```
int **ppi = &pi;
```

指针变量ppi的内容就是指针变量pi的起始地址。于是.....

ppi的值是多少呢？——比如10000。是pi的址。

*ppi的值是多少呢——2006,即pi的值,也是i的址。

**ppi的值是多少呢？——20,即i的值,也是*pi的值。

它可以指向指针，亦可以指向指针数组。

指向指针的指针应用实例

设计一个函数: `void find1(char array[], char search, char * pa)`

要求: 这个函数参数中的数组`array`是以`\0`值为结束的字符串, 要求在字符串`array`中查找与参数`search`给出的字符相同的字符。如果找到, 通过第三个参数(`pa`)返回`array`字符串中首先碰到的字符的地址。如果没找到, 则为`pa`为`NULL`。

依题意, 实现代码如下。

```
void find1(char [] array, char search, char * pa)
{
    int i;
    for (i=0;*(array+i)!='\0';i++)
    {
        if (*(array+i)==search)
        {
            pa=array+i // pa得到某元素的地址
            break;
        }
        else if (*(array+i)==0)
        {
            pa=0;
            break;
        }
    }
}
```

你觉得这个函数能实现所要求的功能吗?

下面调用这个函数试试。

```
void main()
```

```
{
```

```
    char str[]="afsd fsdfdf"; //待查找的字符串
```

```
    char a='d'; //设置要查找的字符
```

```
    char * p=0; //
```

```
    find1(str,a,p); //调用函数以实现查找操作。
```

```
    if (0==p )
```

```
    {
```

```
        printf ("没找到! \n"); //如果没
```

```
    }
```

```
    else
```

```
    {
```

```
        printf("找到了, p=%d",p); //如果找到则输出此句
```

```
    }
```

```
}
```

ch值为' d'，而str字符串的第四个字符就是' d'，应该找得到呀！为何没找到！

上面代码，你认为会输出什么呢？

分析：先看函数定义处：

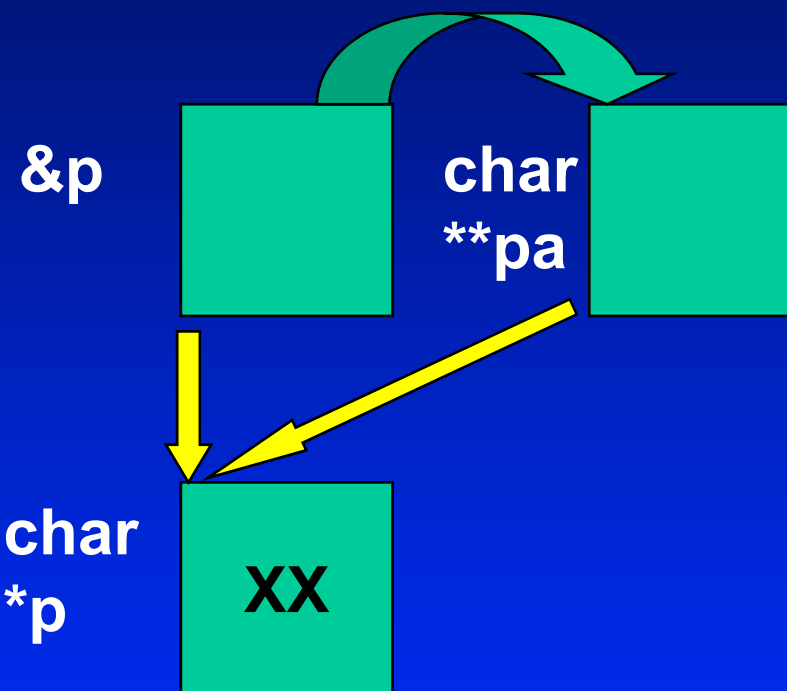
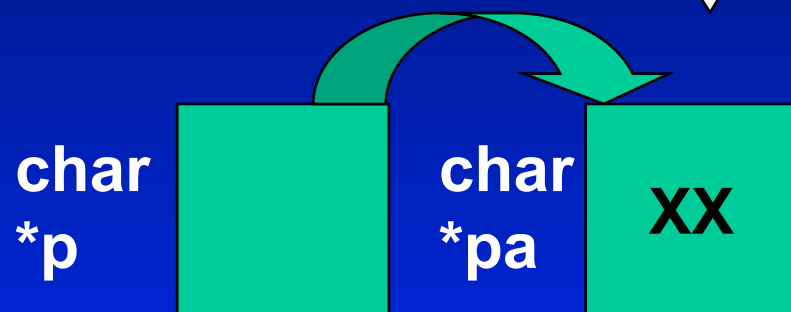
void find1(char [] array, char search, char * pa)

再看调用处：**find1(str,a,p);**

请仔细考虑此时形实结合所发生的事：**array**得到了数组名**str**，**search**得到了**a**的值，**pa**得到了**p**的值（而非**p**的地址）！但**p**并未得到**pa**的值（某元素的地址）。

可见尽管使用了指针，也并没实现传址，当实参形参都是指针时，它们也仅仅是传值——传了别人的地址，没有传回来。 画出内存使用图示就清楚了。

pa=...就是更改形参pa的值，而p的只并没有被修改。



*pa=...就是更改形参pa所指向的内容的值，就是修改了p的值。

修正:

```
void find2(char [] array, char search, char ** ppa)  
{  
    int i;  
    for (i=0;*(array+i)!=0;i++)  
    {  
        if (*(array+i)==search)  
        {  
            *ppa=array+i  
            break;  
        }  
        else if (*(array+i)==0)  
        {  
            *ppa=NULL;  
            break;  
        }  
    }  
}
```

主函数的调用处改如下：

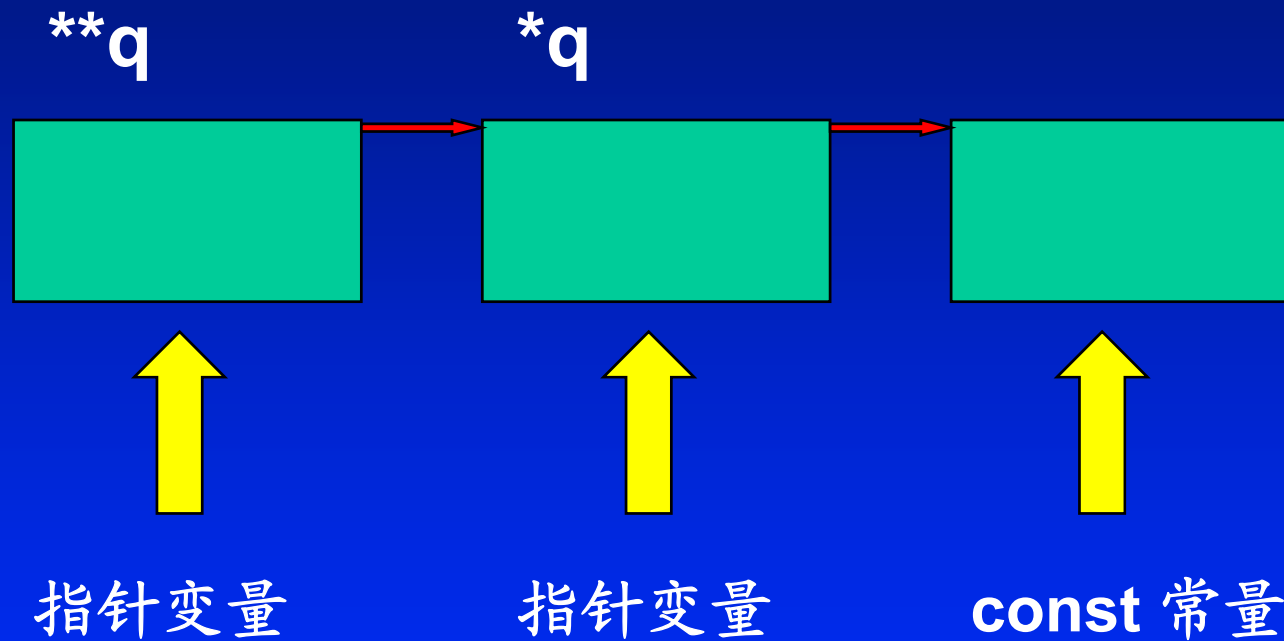
find2(str,a,&p); //调用函数以实现操作。
这样形实结合所发生的操作如下：

ppa=&p;

ppa是指向指针**p**的指针。
对***ppa**的修改就是对指针**p**的修改。

看懂了这个例子，也就完全掌握了指针的精华。

如果定义了: `const int **q;`



类型是 `const int *`

一个有趣的实例

不要将指针的用法照搬到指向指针的指针头上:

`char * pa` 和 `const char * pa` 是类型相容的;

但 `char ** pa` 和 `const char ** pa` 则类型不相容!

看下面代码:

```
#include<stdio.h>
```

```
void fun1(const char *p) { }
```

```
void fun2( const char **p) { }
```

```
void main()
```

```
{
```

```
    char *p1;
```

```
    fun1(p1); // 非常指针作实参, OK!
```

```
    char **p3 = &p1;
```

```
    // fun2(p3); // 非常指向指针的指针针作实参, error
```

```
}
```

这是因为：

当 `char *p1;` `const char *p2;` 时

`p2 = p1;` 合法。但 `p1 = p2;` 就非法。

对于 `char ** p3`；它是指向非常量的二级指针（它所指的指针也指向非常量：`char * q1; p3 = &q1;`）；

而 `const char ** p4`；则是指向常量的二级指针（即它所指的指针是指向常量的：`const char * q; p4 = &q;`）也就是说，`char *` 和 `const char *` 类型不相容（这不同于 `char` 和 `const char` 类型）！所以 `p4 = p3` 是非法的，当然 `p3 = p4` 也是非法的。

void 类型的指针

又称为“万能型”指针。

所谓“万能”是说该指针没保存类型信息，只是保存了个地址，不知该咋用这个地址。

还记得一个故事说的，某人奉命送一封信，交给对方后，对方掏出枪指着他的脑门说，你知道信上说什么吗？让我杀掉来人。

void类型的指针就像这个送信人：只管送信，不知信上说的是什麼。 void类型的指针就这么傻！！

于是使用时一定要强制类型转换——对傻瓜的使用。

void类型的指针(**void * pv;**), 是表示行为不确定的指针, 但这不影响它已经被分配了空间, 甚至得到了值——别人的地址。

void类型的指针不能参与算术运算 (如 **++**, **--**), 不能参与求值运算(***pv**), 只能进行被赋值、比较、**sizeof()**操作。

void 指针的操作

1. 可以与另一个任何类型的指针相比较;
2. 可以向函数的void类型的形参传入指针, 但不可向非void类型的形参传入void指针;
3. 函数可以返回void类型的指针;
4. 可以向另一个该类型的指针赋值;
5. 可以个到任何类型的指针的值;
6. 不得对其所指向的对象进行任何操作。

指针使用过程中需要注意的问题

- 未分配内存就使用它;
- 内存分配未成功, 却使用了它;
- 内存分配虽然成功, 但是尚未初始化就使用它;
- 内存分配成功并且已经初始化, 但操作越过了边界;
- 忘记了释放内存或只释放一部分, 造成内存泄露;
- 释放了内存却继续使用它;
- 程序中的对象调用关系过于复杂, 实在难以搞清楚某个对象究竟是否已经释放;
- 函数的**return**语句写错了, 返回了栈内存的地址;
- 使用**free**或**delete**释放了内存后, 没有将指针设置为**NULL**;
- 释放的顺序不对。

常见的错误举例

没有申请空间就使用

```
f()
```

```
{ int* p;
```

```
  *p = 0; // 1. 压根没申请空间就使用
```

```
}
```

数组或指针越界

```
void main()
```

```
{
```

```
  char *a, *b;
```

```
  a = new char[10]; // 2. 申请了但不见得成功
```

```
  b = a + 20;
```

```
  *b = 0; // 3. 指针超出了申请空间的范围
```

```
}
```

这是典型的新手错误。
他没意识到内存分配会
不成功。

函数结束时忘记释放，导致内存泄漏

```
void gimme()
```

```
{
```

```
    char *p;
```

```
    p = (char *)malloc(10);
```

```
    return; // 没有释放p的空间就返回了
```

```
}
```

```
void main()
```

```
{
```

```
    gimme();
```

```
    return;
```

```
}
```

释放内存顺序不对，导致内存释放了还在使用它

```
typedef struct ptrblock
```

```
{   char *ptr;
```

```
} PB;
```

```
void main()
```

```
{
```

```
    PB *p;
```

```
    p = (PB )malloc(sizeof(PB)); // 申请了PB变量
```

```
    p->ptr =(char*) malloc(10);
```

```
    ...
```

```
    free(p);
```

```
    free(p->ptr ); // p的空间已经释放了，还使用它
```

```
}
```

内存已释放，指向它的指针还在使用

```
char foo(char *p)
{
    return *(p+1);
}
main()
{
    char *a;
    a = malloc(10);
    free(a);
    foo(a);
    return (0);
}
```

常见的错误举例——思考下面程序的运行结果

```
void main()
{
    char *input1;
    char *input2;
    input1 = (char*)malloc(20);
    strcpy (input1, "this is string1");
    printf("%s\n", input1);
    free(input1);
    input2 = (char*)malloc(20);
    strcpy (input2, "this is string2");
    printf("%s\n", input2);
    if( input1 != NULL )
        strcpy (input1, "hello world");
        printf("%s\n", input1);
    printf("%s\n", input2);
}
```


问题表(1)

- Q1:下面这段程序的执行结果是什么?

```
char *GetString(void)
{
    char p[] = "Hello World";
    return p;
}

void Test(void)
{
    char *str = NULL;
    str = GetString();
    printf("%s\n", str);
}
```

- Q2:char a[] = "Hello World"; 与 char *p = "Hello World"; 有区别吗?

a[0] = 'Y'; 能成功吗?

p[0] = 'Y' 能成功吗?

问题表 (2)

➤ Q3:这段代码有问题吗?

```
char *p = (char *)malloc(100);
```

```
strcpy( p, "hello" );
```

```
free(p);
```

```
...
```

```
if (p != NULL)
```

```
{
```

```
    strcpy(p, "world");
```

```
}
```

问题表 (3)

➤ Q4:这段代码有问题吗?

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#define SIZE 4
```

```
void main()
{ int i,j;
  char str[100];
  char *tstr[SIZE];           //创建指针数组
  char *tp=NULL;
  printf("请输入字符串，以回车间隔!\n");
```

```
for(i=0;i<SIZE;i++)
{ scanf("%s",str);
  tstr[i]=(char *)malloc(strlen(str));//指针必须申请空间。
  strcpy(tstr[i],str);
}
printf("\n");
for(i=0;i<SIZE-1;i++)
  for(j=i;j<SIZE;j++)
    if(strcmp(tstr[i],tstr[j])>0)
    {      tp=tstr[i];
      tstr[i]=tstr[j];
      tstr[j]=tp;
    }
for(i=0;i<SIZE;i++)
{  printf("%s\n",tstr[i]);
  free(tstr[i]);
}
}
```

千万记住+1